Compilation – TD2 : Activations – Implémentation

Nawfal ‘Massine’ MALKI – 4991 – STI 3A TD3

Question 1 : inspecter yarser.ypp et compléter le tableau suivant :

Non-terminal/règle	**Catégorie syntaxique**	**Exemple**
Function	Déclaration de fonction	Int fact (int n) {...}
declare_args	Déclaration d’arguments dans un prototype de fonctions	int x,y;

declare_local_vars	Déclaration de variables locales dans le corps d’une fct	Int pow(int x, int n){
		int result ;
		...
		}

Règle l.123	Appel à une fonction	Pow(5,2);
Caller_arg_list	Les arguments d’une fonction peuvent-être des expression	Pow(2+1,1) ;

Règle l.200	Retour d’une fonction	Int foo(){
		…
		return bar ;
		}

function_list	Une fonction comprend une liste de fonction	Void main(){
		...
		x = pow(5,2);
		y = f(x);
		...
		}

prog	Programme composé de fonctions dont au moins une s’intitulant main()	Double hypothenuse (double a, double b) {
		...
		}
		…
		svoid main(){
		...
		}

Question 2 : Compilation des appels de fonctions

1. Comment les arguments sont poussés sur la pile ?

La fonction appelante empile les arguments de la fonction appelée un par un en commençant par le plus à gauche.

2. Ajouter le call l.123

```
| TK_ID TK_LPAR caller_arg_list TK_RPAR { call($1) ; }
```

3. Où se trouve le résultat ?

Au moment du return, le résultat à retourner se trouve à la tête de la pile, juste avant @retour. En effet la fonction iret n (n le nombre d'arguments) dépile le résultat, désalloue les variables locales à la fonction puis empile le résultat avant de jump à @retour.

4. Test sur call.c

```
[compilation@ubuntu test]$ cat call.c
int f(int p, int q)
{
  return p+q;
}

void main()
{
  int x;
  int y;
  x = f(x,x+y);
}
[compilation@ubuntu test]$ ../bin/scc call.c
        call    main
        mov     rax,0
        ret     0
        push    -4
        push    -3
        add
        push    0
        push    0
        push    1
        add
        call    f
        pop     0
[compilation@ubuntu test]$
```

Question 3 : Compilation des fonctions

1. Quelle est l'action de declare_args et declare_local_vars ?

Declare_args permet de déclarer les arguments d'un prototype de fonction. Declare_local_vars permet de déclarer les variables locales d'une fonction.

2. Compléter les parties manquantes

```
function:
  type TK_ID TK_LPAR { reset_symbol_table(); } declare_args TK_RPAR
  TK_LACC declare_local_vars
{
  /*
    Après l'accolade ouvrante, on définit un label (au cas où la fonction serait
    récursive) pour la fonction définie par TK_ID ($2) et on alloue le nombre de
    variables locales.
  */
  label($2);
  alloc(get_nb_local());
}
  stmt_list
{
  /*
    Après la liste des statements et avant l'accolade fermante du corps d'une
    fonction, il faut désallouer l'espace alloué pour les variables locales et
    sauter à @retour. On utilise pour cela les instructions free() puis ret().

    Note : on gère ici un cas général où une valeur de retour n'est pas
    forcément spécifiée. Après une liste de statements, l'issue nominale d'une
    fonction est un jump vers @retour. Le traitement du cas où il y aurait une
    valeur de retour (iret) se fera en tant qu'une statement spécifique dans
    stmt_list. Étant donné qu'iret() s'occupe à la fois de désallouer les
    variables déclarées, de remonter la valeur de retour dans la pile et de
    jump vers @retour, la fonction ret() ci-dessous ne posera pas problème,
    elle sera court-circuitée de toute manière.
  */
  free();
  ret(get_nb_argument());
}
  TK_RACC
;
```

3. Test sur call.c

```
[compilation@ubuntu test]$ ../bin/scc call.c
        call    main
        mov     rax,0
        ret     0
f:
        alloc   0
        push    -4
        push    -3
        add
        free
        ret     2
main:
        alloc   2
        push    0
        push    0
        push    1
        add
        call    f
        pop     0
        free
        ret     0
```

Question 4 : Compilation du return

1. Rappeler l'instruction à utiliser pour le return. Chercher dans Backend.h comment la produire.

Pour le return, on utilise l'instruction iret() donc le prototype dans Backend.h est void iret(int n) avec n le nombre d'arguments alloués pour la fonction. On peut obtenir ce nombre d'arguments avec la fonction get_nb_arguments() déjà implémentée dans SymbolTable.cc.

2. Comment l'expression à retourner est-elle poussée sur la pile ?

La fonction iret() dépile le résultat, désalloue l'enregistrement d'activation (n arguments), empile le résultat et saute à @retour.

3. Compléter la règle.

```
| TK_RETURN expr TK_SEMI { iret(get_nb_argument());  }
```

4. Test sur call.c

```
[compilation@ubuntu test]$ ../bin/scc call.c
        call    main
        mov     rax,0
        ret     0
f:
        alloc   0
        push    -4
        push    -3
        add
        free
        ret     2
main:
        alloc   2
        push    0
        push    0
        push    1
        add
        call    f
        pop     0
        free
        ret     0
```

Question 5 : Validation du compilateur

1. Substituer Backend-stack.o par Backend-x86.o dans le Makefile
2. Recompiler avec make
3. Tester sur call.c (ajouter des input() et des print()), fact.c, fib.c et pgcd.c.

```
[compilation@ubuntu test]$ make pgcd.out
../bin/scc pgcd.c
nasm -f elf64 pgcd.asm
gcc -no-pie -o pgcd.out pgcd.o
rm pgcd.asm pgcd.o
[compilation@ubuntu test]$ ./pgcd.out
21
3
3
[compilation@ubuntu test]$ make fib.out
../bin/scc fib.c
nasm -f elf64 fib.asm
gcc -no-pie -o fib.out fib.o
rm fib.asm fib.o
[compilation@ubuntu test]$ ./fib.out
13
[compilation@ubuntu test]$ make fact.out
../bin/scc fact.c
nasm -f elf64 fact.asm
gcc -no-pie -o fact.out fact.o
rm fact.asm fact.o
[compilation@ubuntu test]$ ./fact.out
6
[compilation@ubuntu test]$
```

Question 6 : Compilation du do/while

1. Proposer un schéma de traduction pour le do/while.

```
[[do/while(C) S]] =
     while :
          [[S]]
          [[C]]
          jz end:
          jmp while
     end:
```

2. Implémenter le do/while dans le compilateur. Tester sur dowhile.c.

```
/* Dans lexer.c, on rajoute le token TK_DO */
"do"                    return TK_DO;

/* On rajoute ce token dans parser.ypp */
%token TK_ASSIGN TK_IF TK_ELSE TK_WHILE TK_DO

/* On implémente le do/while */

| TK_DO
    {
      string label_while = new_label("while");
      label(label_while);
      push_label(label_while);
    }
  stmt_block
  TK_WHILE test TK_SEMI
    {
      string label_endwhile = new_label("endwhile");
      jz(label_endwhile);
      string label_while = pop_label();
      label(label_endwhile);
    }
```